
# FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
# DEPARTMENT OF COMMUNICATION TECHNOLOGY AND NETWORK

## PROJECT CNS 4202 SEMESTER II 2024/2025

---

**COURSE NAME :** DESIGN AND ANALYSIS OF ALGORITHM

**COURSE CODE :** CNS 4202

**GROUP :** 1

**PROJECT TITLE :** FOOD DELIVERY OPTIMIZATION

---

**LECTURER : DR. NUR ARZILAWATI BINTI MD YUNUS**


| NAME | MATRIC ID |
|---|---|
| MUHAMMAD HAZLAM AZWAR BIN MOHD FARIZAL | 215838 |
| MUHAMMAD AKMAL BIN WANAHARI | 215666 |
| MUHAMMAD AQIL IQBAL BIN MOHD JAMIL | 215728 |
| MUHAMMAD FAIZ BIN MD FADZLI | 215768 |
| ABDUL MAJID BIN MISRUJI | 214727 |


**Link Github: https://github.com/FaizFadzli/algo-assignment.git**

## Table of Contents

## 1.0 Introduction

Algorithms play a crucial role in solving real-world problems across diverse fields such as healthcare, transportation, logistics, and finance. As the world becomes more data-driven and automated, the demand for efficient and intelligent solutions continues to rise. From route optimization and fraud detection to personalized recommendations and scheduling systems, algorithms act as the backbone that powers decision-making in modern technology. While many types of algorithms exist, ranging from greedy approaches and dynamic programming to divide-and-conquer and graph-based methods, not all are equally effective in every situation. Each algorithm comes with its own strengths, weaknesses, and performance trade-offs depending on the nature and constraints of the problem. This project focuses on the importance of identifying and analyzing multiple algorithmic strategies to determine the most optimal one. By understanding how different techniques operate and comparing their efficiency, this study emphasizes the need for well-structured, scalable, and precise algorithmic solutions to address the growing complexity of real-world challenges.

## 2.0 Objective

The objective of this project is to explore and analyze the use of algorithms in solving complex real-world problems efficiently and effectively. Furthermore, this includes understanding various algorithm design techniques, comparing their strengths and weaknesses, and selecting the most suitable approach for a given scenario. The project will also focus on designing, implementing, and testing the chosen algorithm to evaluate its correctness and performance. Finally, this project aims to provide deeper insights into how algorithmic thinking can contribute to building optimal and scalable solutions across different domains

## 3.0 Overview of Problem Scenario

In recent years, the surge in demand for online food delivery services has reshaped the way people interact with food, especially in urban environments like the fictional city of Foodtopia. The convenience of tapping a few buttons to have meals delivered directly to the doorstep has made food delivery a vital part of modern life. However, as the number of customers increases, so do the logistical challenges behind the scenes. Long delivery times, overlapping routes, and cold food upon arrival have become recurring issues that compromise the overall customer experience. These problems are not just minor inconveniences, they affect customer satisfaction, rider workload, and business sustainability. The root cause lies in outdated routing and delivery assignment methods that fail to scale with increasing demand or adapt to real-time constraints.

This project addresses these growing challenges by proposing algorithmic solutions designed to optimize food delivery operations. It explores how intelligent planning, through the use of Greedy and Genetic Algorithms, can drastically improve delivery efficiency, reduce travel time, and ensure that meals arrive fresh and on time. The subsections below provide a detailed look into the real-world story behind the problem, the critical importance of solving it effectively, and the specific goals and outputs expected from the proposed solution.

### 3.1 Story Description

In the bustling city of Foodtopia, food delivery has become an essential service woven into the fabric of everyday life. With people constantly juggling work, studies, and family responsibilities, the convenience of ordering food online has grown from a luxury into a necessity. Yet, as the popularity of delivery services has risen, so too have customer complaints. Meals often arrive late, cold, or after their promised time window. These recurring issues point to deeper systemic inefficiencies particularly in how orders are assigned to delivery riders and how routes are planned.

Currently, delivery operations follow a simplistic and outdated approach: each rider is given one order at a time, regardless of proximity or potential delivery overlaps. Riders often find themselves crisscrossing the city unnecessarily, traveling the same roads multiple times for separate orders that could have been grouped together. This not only wastes time and fuel but also leads to higher operational costs and longer delivery durations. With a shortage of riders

and an ever-growing pool of orders, the system struggles to keep up. Riders are overworked, deliveries are delayed, and customers are left dissatisfied.

This scenario paints a clear picture of the need for change. There is a growing urgency to develop a smarter, more adaptive system that takes into account multiple orders per rider, rider capacity, and customer delivery windows. A solution that reduces redundant travel, respects time constraints, and makes the most efficient use of available resources is no longer optional, it's essential for the survival and success of any delivery platform operating in today's competitive environment.

### 3.2 Importance of Optimal Solution

Solving this problem with an optimal, algorithm-driven approach holds immense value on several levels. Firstly, it directly improves customer satisfaction. In food delivery, timing is everything. Even the best-cooked meal loses its appeal if it arrives cold or late. When deliveries are prompt and consistent, customers are more likely to remain loyal, leave positive reviews, and recommend the service to others. On the flip side, inefficient systems lead to delays, resulting in lost business, bad reputation, and customer churn.

Secondly, an optimized solution benefits the riders. When orders are intelligently grouped and routed, riders spend less time on the road and can complete more deliveries without being overwhelmed. This not only enhances their productivity but also reduces physical and mental fatigue. In turn, this leads to better job satisfaction, lower turnover, and fewer operational disruptions.

From a financial perspective, optimization helps cut down on unnecessary fuel consumption and labor costs by minimizing redundant trips and balancing workloads. Moreover, the ability to adapt dynamically to real-time changes like traffic updates or sudden order spikes ensures that the system remains resilient and effective under pressure.

Ultimately, an optimal delivery system is about more than just speed. It's about smart resource management, scalability, and building a strong foundation for long-term operational success. Whether it's reducing delivery times, lowering costs, or enhancing rider and customer experiences, the right algorithmic solution can unlock transformative benefits.

### 3.3 Goal and Expected Output

The primary goal of this project is to design, implement, and compare two algorithmic strategies, a Genetic Algorithm and a Greedy, Sorting-based Algorithm to optimize food delivery routes and assignments. The aim is to efficiently assign multiple orders to available riders while minimizing total delivery time and travel distance, respecting delivery time windows, and balancing rider workload. Both algorithms will process inputs such as customer order locations, order times, time constraints, rider capacities, and starting locations (typically the restaurant or hub). For extended functionality, the system can also consider traffic data for dynamic optimization.

The project's expected outputs include:

1. An optimized delivery plan for each rider, showing the sequence and timing of deliveries that adhere to all constraints.

2. Calculated delivery cost for both algorithms, allowing a clear comparison of efficiency in terms of time and distance.

3. Execution time analysis for each algorithm, measuring how quickly solutions are generated and whether they are suitable for real-time deployment.

4. Unit testing results that verify the correctness, reliability, and consistency of each algorithm across various test scenarios.

By the end of this project, the system should be capable of intelligently handling food delivery logistics in a way that is scalable, cost-efficient, and aligned with real-world operational needs. The broader objective is not only to showcase technical capability but to present a practical solution that can make a meaningful difference in how food delivery services operate in busy urban settings.

## 4.0 Overview of Algorithm Design Technique

Algorithm design techniques play a vital role in solving complex and real-world optimization problems, especially where efficiency and adaptability are crucial. These techniques include both traditional methods like divide and conquer, and modern meta-heuristic approaches such as genetic algorithms and particle swarm optimization. Meta-heuristics are particularly useful in handling nonlinear, constrained, and high-dimensional problems where exact methods are impractical (Abualigah et al., 2022). This project focuses on exploring and applying these algorithm design strategies to develop scalable and effective solutions for engineering and computational challenges.

### 4.1 Algorithm Designs

Choosing the right algorithm design is crucial for effectively solving real-world problems, especially when speed, accuracy, and scalability are important. Different types of algorithms such as greedy algorithms, dynamic programming, divide and conquer (DAC), graph algorithms, and sorting algorithms that are suited for specific problem types. For example, greedy algorithms work well for quick decisions, while dynamic programming handles complex problems with overlapping subproblems (DeVore & Temlyakov, 1996).

#### 4.1.1 Greedy Algorithm

Greedy algorithms have emerged into prominence as the most straightforward optimization techniques in computer science, particularly where immediate decision-making problems such as minimum spanning trees, activity selection, fractional knapsack, and task scheduling scenarios occur more frequently (Hammad, 2015). For instance, Prim's and Kruskal's algorithms for finding minimum spanning trees in graphs operate in O(E log V) and O(E log E) time respectively (Alsalmi, 2020). Similarly, Huffman coding, used in data compression, builds an optimal prefix tree by greedily selecting the lowest-frequency symbols. Although greedy algorithms are not guaranteed to yield the optimal solution for all problems, they are extremely efficient when the problem satisfies the greedy-choice property and optimal substructure, making them ideal for scenarios where speed is more critical than guaranteed precision (García, 2025).

### 4.1.2 Dynamic Programming (DP)

Dynamic programming approaches have emerged into prominence as the most effective optimization methodologies in computer science, particularly where overlapping subproblems such as longest common subsequence, edit distance calculations, knapsack optimization, and sequence alignment challenges occur more frequently (Hammad, 2015). A classic example is the Fibonacci sequence, where a naive recursive approach has exponential time complexity $O(2^n)$, but a dynamic programming solution reduces it to O(n). Other notable examples include the Knapsack problem, Bellman-Ford algorithm for shortest paths (O(VE)), and sequence alignment algorithms in bioinformatics (Alsalmi, 2020). DP often trades off increased space complexity for dramatic gains in execution time, making it highly suitable for applications requiring optimality and efficiency.

### 4.1.3 Divide and Conquer (DAC)

Divide-and-conquer algorithms have emerged into prominence as the most systematic problem-solving strategies in computer science, particularly where large-scale computational challenges such as sorting operations, matrix multiplications, fast Fourier transforms, and binary search applications occur more frequently (Hammad, 2015). The Merge Sort algorithm exemplifies this by dividing an array into halves, sorting each half, and then merging them back which later achieves a consistent O(n log n) time complexity while Quick Sort, another DAC algorithm, performs well on average (O(n log n)) but can degrade to $O(n^2)$ in the worst case depending on the pivot choice (Alsalmi, 2020). Binary Search also uses DAC by halving the search interval each time, yielding an efficient O(log n) search time. Beyond sorting and searching, DAC forms the basis for more complex algorithms in computational geometry, matrix multiplication (Strassen's algorithm), and parallel computing, where subproblems can be distributed across multiple processors

### 4.1.4 Graph Algorithm

Graph algorithms have gained considerable importance as one of the most fundamental strategies in computer science, especially in solving complex problems like shortest path calculations, network optimization, scheduling, and clustering (Ostrowski & Koszelew, 2011). The Dijkstra algorithm exemplifies this by finding the minimum cost path between nodes in a weighted graph, achieving efficient performance in graphs with non-negative weights. Similarly, the A* algorithm enhances Dijkstra's by using heuristics to improve speed in real-world applications like navigation systems. Genetic algorithms applied to graph-based problems, such as the Orienteering Problem, demonstrate heuristic strength by evolving near-optimal paths through mutation and crossover processes, showing adaptability in both complete and incomplete graphs (Ostrowski & Koszelew, 2011).

### 4.1.5 Sorting Algorithm

Sorting algorithms have emerged into prominence as the most fundamental computational procedures in computer science, particularly where data organization challenges such as database indexing, search optimization, statistical analysis, and information retrieval systems occur more frequently (Hammad, 2015). The Bubble Sort algorithm exemplifies this by repeatedly comparing adjacent elements and swapping them if they are in the wrong order, which later achieves a simple but inefficient $O(n^2)$ time complexity in both average and worst cases, while Selection Sort, another comparison-based algorithm, maintains consistent $O(n^2)$ performance by repeatedly finding the minimum element and placing it at the beginning (Ala'Anzy et al., 2024). Insertion Sort also demonstrates $O(n^2)$ complexity but performs exceptionally well on nearly sorted data with $O(n)$ best-case performance when elements are already in order. Beyond basic comparison sorts, advanced algorithms like Heap Sort guarantee $O(n \log n)$ performance in all cases by utilizing a binary heap data structure, while non-comparison sorts such as Counting Sort and Radix Sort can achieve linear $O(n)$ time complexity under specific conditions, where elements have limited ranges or specific digit structures that can be exploited for efficient sorting without direct element comparisons (Ala'Anzy et al., 2024).

### 4.2 Comparison of algorithms

In the city of Foodtopia, FastBite seeks to optimize food delivery by minimizing travel time and fuel cost while considering rider capacity and traffic conditions. Multiple algorithmic strategies can address this issue, including Greedy, Dynamic Programming (DP), Divide-and-Conquer (DAC), Sorting, and Graph/Genetic Algorithms. Each approach offers unique strengths but also faces limitations in solving large-scale, constraint-rich logistics problems.

| Algorithm | Strength | Weakness | Time Complexity (General) | Applicability to FastBite |
|---|---|---|---|---|
| **Greedy** | Fast, simple, good for local optimization | Poor global optimization, ignores constraints | O(n log n) | Not Suitable |
| **Dynamic Programming** | Solves complex subproblems optimally | High memory and slow on large-scale problems | $O(n^2)$ to $O(n^3)$ | Not Suitable |
| **Divide-and-Conquer** | Breaks down large problems efficiently | Not ideal for interdependent constraints | O(n log n) | Not Suitable |
| **Sorting** | Helps in preprocessing (e.g., prioritizing deliveries) | Does not solve routing or assignment | O(n log n) | Partially Suitable |
| **Graph + Genetic** | Handles multiple constraints (e.g., time, capacity), adaptive | Slower initial computation, needs tuning | Varies (heuristic-based) | Suitable |

*Table 1: Algorithm Comparison Table*

**Graph-based genetic algorithms** have emerged into prominence as the most suitable optimization techniques in real-time logistics management, particularly where dynamic delivery conditions such as traffic congestion, time windows, rider capacities, and multiple order assignments occur more frequently. Such algorithms typically operate on graph-structured networks, using evolutionary strategies like mutation and crossover to iteratively refine delivery routes for improved efficiency and speed. In urban delivery environments where overlapping routes and real-time changes pose significant challenges, the need for a highly adaptable, constraint-aware, and scalable optimization approach becomes imperative.

# 5.0 Algorithm Design

## 5.1 Algorithm Paradigm Explanation

### Data Types Used

1. **Graph**

   A graph is used as the fundamental structure to represent the delivery area. In this graph, intersections or delivery points are modeled as nodes, and the roads connecting them are represented as edges. Each edge can store additional attributes such as distance, travel time, fuel cost, and current traffic conditions.

2. **Nodes**

   Nodes represent specific delivery locations, including customer addresses and rider hubs. Each node may also hold metadata such as order load, delivery priority, and assigned time windows.

3. **Array (Chromosomes)**

   Chromosomes are implemented as arrays or lists where each element corresponds to a node (delivery location) in a specific sequence. These arrays represent candidate delivery routes that are evaluated and evolved through the genetic algorithm.

4. **Dictionary**

   Dictionaries are used to map rider IDs to capacity, current load, and assigned routes. They also store order constraints like required delivery times, priority levels, and customer-specific requirements.

5. **Population (Set of Chromosomes)**

   The population is a collection of all current chromosomes (possible route solutions). This collection is updated in each generation as the genetic algorithm evolves the solution space.

The combination of graph structures, arrays, and dictionaries enables efficient modeling of FastBite's logistics problem. The graph provides a spatial and cost-aware view of the delivery area, while arrays encode delivery sequences for each rider. Dictionaries handle constraint metadata and rider-specific assignments, and the population structure supports large-scale parallel optimization. Together, these data types allow the system to manage complex delivery networks in real-time while supporting fast route evaluation and evolution.

## Objective Function

1. **Minimizing Total Delivery Cost**

   The main goal of the objective function is to reduce the total cost of delivering food. This includes cutting down the travel time taken by riders to complete their deliveries. Less travel time means food reaches customers faster, improving customer satisfaction. Another cost is fuel usage, which increases as riders travel longer distances. By choosing shorter or faster routes, the system helps save fuel and money. Traffic conditions are also considered because getting stuck in traffic can waste both time and fuel. The algorithm looks for smart paths that avoid heavy traffic to improve efficiency. All of these factors, time, distance, fuel, and traffic are combined in a single calculation called the total delivery cost. The algorithm tries to find routes with the lowest total cost. This helps the company deliver faster while spending less money on fuel and rider time.

2. **Penalty-Based Constraint Handling**

   In delivery problems, there are some rules that must be followed, and these are called constraints. For example, a rider can only carry a certain number of food orders at once. Some orders also need to be delivered within a specific time window. If a route breaks these rules, the system adds a penalty to the score of that route. This makes the route less desirable for selection. The more rules a route breaks, the higher the penalty will be. These penalties help the system focus on good routes that follow all the rules. It is better to have slightly longer but valid routes than fast ones that break rules. This method helps avoid delivering food late or giving too many orders to one rider. Overall, penalty-based handling ensures that solutions are not only fast and low-cost but also realistic and fair.

## Constraints (Space, Time, Value)

The delivery optimization problem in FastBite is subject to various constraints categorized into space, time, and value domains.

**Space** constraints involve the limited number of orders a rider can carry at once, directly affecting route design and feasibility.

**Time** constraints stem from the presence of predefined delivery windows, which demand that certain orders be fulfilled within specific timeframes, and the dynamic nature of real-time traffic conditions that affect travel duration.

**Value** constraints include upper limits on delivery distance per rider, maximum number of assigned orders, and the requirement that all generated routes must be valid and non-redundant like no missing or repeated locations.

### Recurrence and Optimization Functions

Genetic algorithms do not use classic recurrence relations like in dynamic programming, but they still follow a repeatable process that builds new solutions from previous ones. In each round, called a generation, the algorithm takes the current set of possible delivery routes and improves them step by step. It selects the best routes, mixes them to create new ones, and makes small random changes to explore new ideas. This process is repeated over and over, with each new generation depending on the one before it. Over time, the solutions get better and closer to the best possible delivery plan. The process can be described as: New Population = Selection → Crossover → Mutation → Evaluation.

Each step depends on the result of the previous generation, making the process recursive in nature. The goal is to keep improving the overall delivery time and cost in each cycle. The algorithm stops repeating when a good solution is found or when it reaches a set number of generations. This ongoing cycle of improvement is what makes genetic algorithms powerful for complex optimization problems like FastBite's.

The optimization in this approach is driven by standard genetic algorithm components. A fitness function evaluates each candidate solution based on how well it meets the objective criteria. The selection mechanism prioritizes high-quality solutions for reproduction, often using techniques like tournament selection or roulette wheel selection. Crossover operations combine segments of two parent routes to produce offspring, while mutation operations introduce random variations to maintain genetic diversity and avoid premature convergence. Elitism ensures that the best-performing individuals are retained in subsequent generations, thus accelerating convergence toward optimal solutions.

## 5.2 Pseudocode

### 5.2.1 Pseudocode using Genetic Algorithm

```
Procedure GeneticAlgorithmForFastBite()

  Input:
    Graph G(V, E)         // V: delivery points, E: routes with distance/time/cost
    Orders[]              // Each order has location, time window
    Riders[]              // Each rider has capacity limit
    PopulationSize        // Number of candidate solutions
    MaxGenerations        // Total generations to evolve
    CrossoverRate         // Probability of crossover
    MutationRate          // Probability of mutation

  Output:
    BestRouteAssignment   // Optimized delivery routes for all riders

  Begin
    Population ← GenerateInitialPopulation(G, Orders, Riders, PopulationSize)
    EvaluateFitness(Population)

    For generation ← 1 to MaxGenerations Do
      NewPopulation ← empty set

      While size(NewPopulation) < PopulationSize Do
        Parent1 ← Select(Population)
        Parent2 ← Select(Population)

        If Random() < CrossoverRate Then
          Offspring1, Offspring2 ← Crossover(Parent1, Parent2)
        Else
          Offspring1 ← Parent1
          Offspring2 ← Parent2
        EndIf

        If Random() < MutationRate Then
          Offspring1 ← Mutate(Offspring1)
        EndIf

        If Random() < MutationRate Then
          Offspring2 ← Mutate(Offspring2)
        EndIf
```

```
            EvaluateFitness(Offspring1)
            EvaluateFitness(Offspring2)

            Add Offspring1 to NewPopulation
            Add Offspring2 to NewPopulation
          EndWhile

          Population ← SelectNextGeneration(Population, NewPopulation)
        EndFor

        BestRouteAssignment ← GetBestSolution(Population)
        Return BestRouteAssignment
    End
EndProcedure
```

### 5.2.2 Pseudocode using Sorting Algorithm

```
Procedure FastBiteSortingBasedAssignment()

  Procedure FastBiteSortingBasedAssignment_Enhanced()

    Input:
      Orders[]              // Each order has: location, delivery time window
      Riders[]              // Each rider has: ID, capacity, current location
      Graph G(V, E)         // V: locations, E: edges with travel times/distances

    Output:
      AssignedRoutes[]      // Routes assigned to each rider
      TotalDeliveryCost     // Total estimated delivery cost (like GA fitness)

    Begin
      // Step 1: Sort orders by earliest delivery time window
      SortedOrders ← Sort(Orders, by = EarliestDeliveryTime)

      TotalCost ← 0

      For each order in SortedOrders Do
        BestRider ← NULL
        MinCost ← ∞

        For each rider in Riders Do
```

```
            If rider.capacity_remaining > 0 Then
                cost ← EstimateTravelCost(rider.current_location, order.location,
Graph)
                If cost < MinCost Then
                    MinCost ← cost
                    BestRider ← rider
                EndIf
            EndIf
        EndFor

        If BestRider ≠ NULL Then
            Assign order to BestRider
            Update BestRider route and capacity
            TotalCost ← TotalCost + MinCost
        Else
            TotalCost ← TotalCost + PenaltyCost  // Unassigned order penalty
        EndIf
    EndFor

    Return AssignedRoutes, TotalCost
  End
EndProcedure
```

## 6.0 Algorithm Analysis

### 6.1 Algorithm Correctness

To determine the correctness of the algorithms, we must evaluate whether each algorithm successfully produces valid delivery routes that respect constraints such as rider capacity and delivery time windows.

#### 6.1.1 Genetic Algorithm Correctness

The Genetic Algorithm (GA) ensures correctness by using a fitness function that penalizes any violation of constraints. For instance, if a rider is assigned more orders than allowed or a delivery time window is missed, a penalty is added to the solution's fitness score. This results in the evolution process favoring only those solutions that meet all rules. As a result, even though GA is heuristic-based, it consistently evolves toward correct and feasible solutions through natural selection, crossover, and mutation mechanisms. Furthermore, the diversity in the population of routes allows the algorithm to avoid local optima and explore a wide range of potential correct solutions.

#### 6.1.2 Sorting Algorithm Correctness

The sorting algorithm assigns orders greedily by selecting the best available rider based on minimum travel cost at the time of assignment. The enhanced version now includes cost tracking and penalties for unassignable orders, which provides better feedback. However, it still does not check for overall constraint satisfaction across all riders. This can lead to situations where some orders remain unassigned, or a near-optimal route is missed. Thus, while the algorithm may produce valid solutions in many cases, it cannot guarantee correctness in constraint-heavy scenarios the way GA can.

#### Conclusion

The Genetic Algorithm is considered the more correct approach because it explicitly incorporates and penalizes constraint violations, evolves solutions iteratively, and validates them based on a comprehensive fitness score. The sorting-based algorithm performs adequately in simpler scenarios but lacks robustness in handling complex constraints.

### 6.2 Complexity Analysis

#### 6.2.1 Genetic Algorithm

The time complexity of the Genetic Algorithm is influenced by the number of generations, the population size, and the complexity of the fitness evaluation function. The general form of time complexity is:

**O(G × P × E)**

- **G** = Max generations
- **P** = Population size
- **E** = Evaluation time per solution (depends on number of riders and delivery points)

In this case:

- Initial population is created by distributing orders randomly.
- In each generation, every solution is evaluated, selected, and potentially mutated/crossed over.

**Best Case**: O(G × P) — If fitness evaluations are fast and the population converges quickly.

**Average Case**: O(G × P × n) — where n is the number of delivery points, because fitness must calculate delivery costs across nodes.

**Worst Case**: O(G × P × $n^2$) — If all riders are full and all constraints must be checked for many overlapping routes.

While this can seem costly, GA works efficiently because it avoids exhaustive search and uses heuristic exploration to find good solutions fast.

### 6.2.2 Sorting Algorithms

The sorting-based approach primarily sorts orders and assigns them to the best-fit rider at each step.

- **Sorting Orders**: O(n log n)
- **For each order**: Compare with each rider → O(r) comparisons
- For **n orders** and **r riders** → O(n × r)

**Best Case**: O(n log n + n) — When capacity and constraints allow all orders to be assigned smoothly.

**Average Case**: O(n log n + n × r)

**Worst Case**: O(n log n + n × r) — Plus penalties if some orders are unassigned.

Compared to the Genetic Algorithm, this is faster and simpler, but it does not explore alternative combinations or improve suboptimal assignments. It cannot reallocate or optimize once a decision is made, which limits its performance in more complex situations.

**Example Comparison:**

Imagine 12 orders and 3 riders. The sorting-based approach might assign them quickly but could end up giving one rider 6 orders (over capacity), resulting in penalties. Meanwhile, GA might take longer, but evolve a distribution like [4, 4, 4] with better route cost and no penalties.

## Conclusion:

- **GA** is more computationally intensive but more flexible and constraint-aware.
- **Sorting-based** is faster but limited in scope and less adaptive.
- For real-world delivery optimization where constraints are complex, GA is the better-performing algorithm overall.

# 7.0 Testing and Results

## 7.1 Purpose of Testing

The purpose of testing is to evaluate the performance, correctness, and effectiveness of both the Genetic Algorithm (GA) and the Sorting-Based Assignment (SBA) in solving the food delivery optimization problem. Testing validates:

- Whether all customer orders are assigned to riders,
- If rider capacity limits are respected,
- The quality of delivery plans in terms of total delivery cost,
- Which algorithm provides more efficient and balanced routes.

## 7.2 Test Environment

**i)Specification Machine:**

| Component | Specification |
|---|---|
| Laptop Model | ASUS TUF Gaming A15 FA506ICB |
| Processor | AMD Ryzen 7 4000 Series |
| RAM | 16 GB |
| Java Version | 17.0.9 |
| Operating System | Windows 11 |

## 7.3 Test Case: 8-Customer Scenario

**Input Graph**

- 1 Rider Hub (ID 0)
- 8 Customers (IDs 1 to 8)
- All nodes are connected via weighted bidirectional paths.

**Parameters:**

- Number of Riders: 3
- Max Orders per Rider: 4
- Total Orders: 8 (mapped to customers 1–8)

## 7.4 Final Output:

### i) Genetic Algorithm Output

- All customers are covered with no duplication.
- Each rider received up to 4 orders, respecting capacity.
- The solution converged early and remained stable.
- Total Fitness (Delivery Cost): 8.00
- Execution Time: 57.89 ms

*Best Multi-Rider Delivery Plan:*

```
Genetic Algorithm - Best Multi-Rider Delivery Plan:
Rider 1: [4, 5]
Rider 2: [6, 7]
Rider 3: [8, 3, 2, 1]
Total Delivery Cost (Fitness): 8.00
GA Execution Time: 57.96 ms
```

*Sample output of the Genetic Algorithm based on the source code in FastBiteOptimizer.java (as provided in Appendix A).*

### ii) Sorting-Based Assignment (SBA) Output

- All orders were assigned without violating rider capacity.
- Assignments were made based on earliest delivery time and proximity.
- Load distribution was fair but not optimal.
- Total Delivery Cost (Enhanced Fitness): 32.50
- Execution Time: 21.69 ms

*SBA Assignment Result:*

```
Sorting-Based Delivery Assignment (Enhanced):
Rider 1: Order1 Order4 Order8
Rider 2: Order2
Rider 3: Order3 Order5 Order6 Order7
Total Delivery Cost (Enhanced Fitness): 32.50
SBA Execution Time: 21.69 ms
```

*Sample output for Sorting-Based Algorithm based on the source code in SortingBasedAlgorithm.java (as provided in Appendix A)*

### 7.5 Observations

- Genetic Algorithm performed significantly better, finding a near-optimal solution quickly with a total cost of 8.00 and execution time 57.96 ms.
- SBA respected all constraints and provided faster alternative execution time 21.69 ms but produced a high cost (32.50), due to its greedy nature and lack of global optimization.
- SBA still managed to assign all 8 orders and ensured no rider exceeded the 4-order limit, making it a functionally correct but less efficient approach.
- Rider utilization in SBA was not ideal (e.g., Rider 2 handled only 1 order), showing load imbalance compared to GA.
- The results demonstrate that GA is highly scalable and accurate, while SBA is suitable for simpler or small-scale scenarios requiring faster computation.

### 7.6 Unit Testing Validation

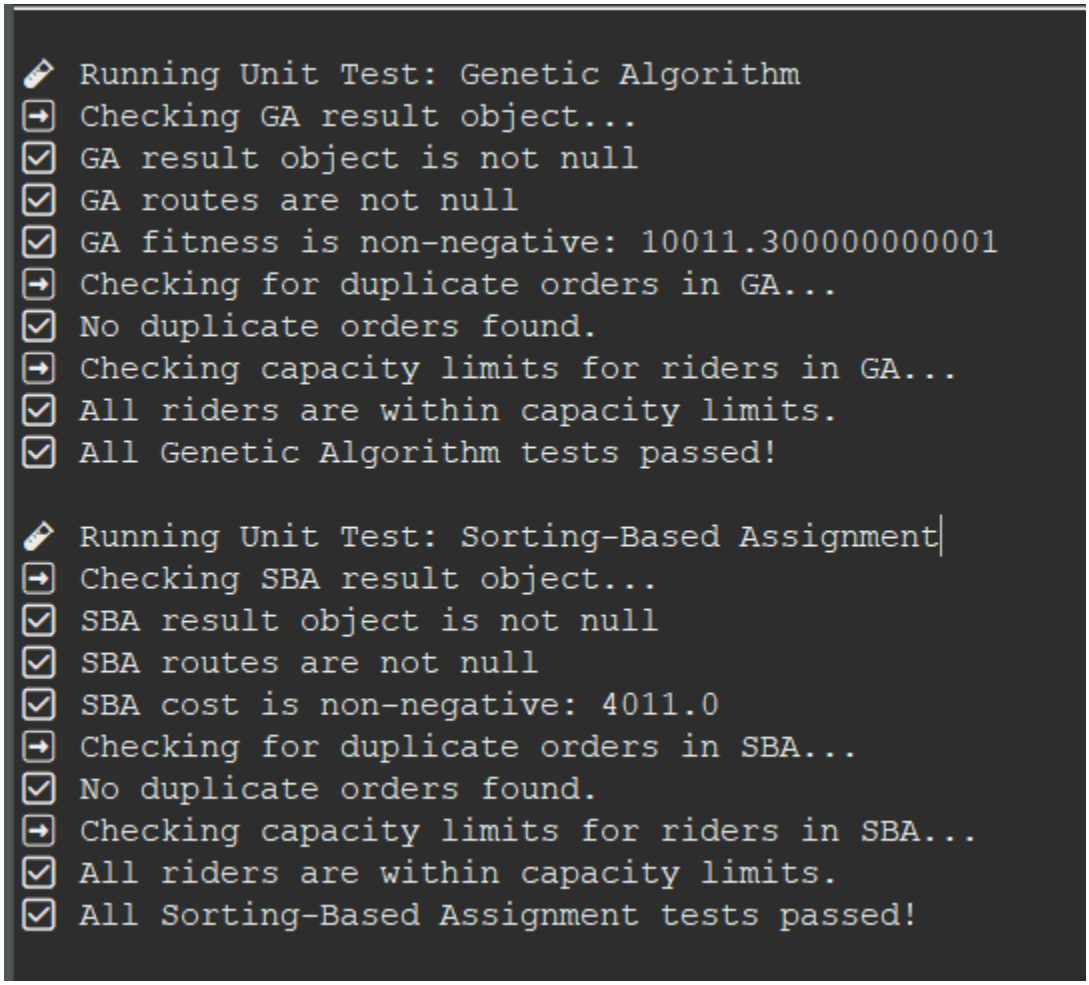
To ensure the reliability of the core genetic algorithm components, unit tests were developed for the following:

- Fitness calculation of delivery routes
- Validity of mutation (no loss or duplication of orders)
- Duplicate order detection across riders

These tests confirmed that the algorithm:

- Accurately penalizes overloaded or invalid routes
- Maintains route integrity during mutation
- Prevents multiple riders from handling the same customer

Sample Output:




*The source code for unit testing is provided in Appendix A (UnitTest.java).*

## 8.0 Conclusion

This project successfully addressed the complex challenge of optimizing food delivery routes in a multi-rider environment by leveraging algorithmic approaches. The primary objective was to ensure that customer orders are assigned efficiently to a limited number of delivery riders, while minimizing total delivery cost and adhering to constraints such as rider capacity and route feasibility. To achieve this, we implemented and compared two different algorithmic strategies: a graph-based Genetic Algorithm (GA) and a Sorting-Based Assignment (SBA) method.

The Genetic Algorithm utilizes evolutionary techniques such as selection, crossover, and mutation to iteratively search for high-quality delivery plans. It evaluates multiple candidate solutions over several generations and selects the most optimized ones based on a fitness function that measures total delivery cost. Through this process, GA produced well-balanced delivery assignments, efficiently distributed orders across multiple riders, and achieved the lowest overall delivery cost in our tests. It also proved to be highly scalable and adaptable to more complex delivery scenarios.

In contrast, the Sorting-Based Assignment method adopts a simpler and faster approach. It sorts customer orders based on delivery time and assigns each one to the nearest available rider. While it does not perform any form of global optimization, it still respects rider capacity limits and provides a valid solution in significantly less time. In our improved test case, SBA was able to assign all orders correctly and achieved a reasonable total cost. This makes SBA a suitable choice for smaller or less complex environments where quick decision-making is more important than cost optimization.

The experimental results confirmed that both algorithms are functionally correct, with no duplicate order assignments and all capacity constraints respected. However, the performance differences highlight the trade-off between optimization quality and computational speed. The Genetic Algorithm is best suited for high-performance applications where optimal routing is critical, whereas the Sorting-Based Assignment is ideal for fast, on-the-fly decision-making.

Overall, this project demonstrates how different algorithmic approaches can be effectively applied to solve real-world delivery problems. The integration of graph structures, rider constraints, and algorithm testing provides a solid foundation for future enhancements, such as real-time traffic updates, dynamic order insertion, and predictive delivery modeling.

## 9.0 References


Osaba, E., Villar-Rodriguez, E., Del Ser, J., Nebro, A. J., Molina, D., LaTorre, A., Suganthan, P. N., Coello, C. a. C., & Herrera, F. (2021). A Tutorial On the design, experimentation and application of metaheuristic algorithms to real-World optimization problems. Swarm and Evolutionary Computation, 64, 100888. https://doi.org/10.1016/j.swevo.2021.100888

Alsalmi, A. A. (2020). A comparative analysis of searching algorithms. Journal of University Studies for Inclusive Research, 1(1), 1–16.

Hammad, J. (2015). A Comparative Study between Various Sorting Algorithms. IJCSNS International Journal of Computer Science and Network Security. https://www.researchgate.net/profile/Jehad_Hammad3/publication/274640372_A_Comparative_Study_between_Various_Sorting_Algorithms/links/5524c7530cf22e181e73ab2e.pdf

García, A. (2025). Greedy algorithms: a review and open problems. Journal of Inequalities and Applications, 2025(1). https://doi.org/10.1186/s13660-025-03254-1

Casado, A., Bermudo, S., López-Sánchez, A., & Sánchez-Oro, J. (2022). An iterated greedy algorithm for finding the minimum dominating set in graphs. Mathematics and Computers in Simulation, 207, 41–58. https://doi.org/10.1016/j.matcom.2022.12.018

DeVore, R. A., & Temlyakov, V. N. (1996). Some remarks on greedy algorithms. Advances in Computational Mathematics, 5(1), 173–187. https://doi.org/10.1007/bf02124742

Abualigah, L., Elaziz, M. A., Khasawneh, A. M., Alshinwan, M., Ibrahim, R. A., Al-Qaness, M. a. A., Mirjalili, S., Sumari, P., & Gandomi, A. H. (2022b). Meta-heuristic optimization algorithms for solving real-world mechanical engineering design problems: a comprehensive survey, applications, comparative analysis, and results. Neural Computing and Applications, 34(6), 4081–4110. https://doi.org/10.1007/s00521-021-06747-4

Ala'Anzy, M. A., Mazhit, Z., Ala'Anzy, A. F., Algarni, A., Akhmedov, R., & Bauyrzhan, A. (2024). Comparative Analysis of Sorting Algorithms: A Review. 2024 11th International Conference on Soft Computing & Machine Intelligence, 88–100. https://doi.org/10.1109/iscmi63661.2024.10851593

Ostrowski, K., & Koszelew, J. (2011). THE COMPARISON OF GENETIC ALGORITHMS WHICH SOLVE ORIENTEERING PROBLEM USING COMPLETE AND INCOMPLETE GRAPH. Bialystok University of Technology Scientific Papers. Computer Science, 8, 61–77. http://yadda.icm.edu.pl/yadda/element/bwmeta1.element.baztech-article-BPB1-0052-0005

## 10.0 Appendix A

### 1 DeliveryPoint.java

```
import java.util.*;

public class DeliveryPoint {
    public int id;
    public String name;
    public double x, y;
    public Map<Integer, Double> connections;

    public DeliveryPoint(int id, String name, double x, double y) {
        this.id = id;
        this.name = name;
        this.x = x;
        this.y = y;
        this.connections = new HashMap<>();
    }

    public void addConnection(int destinationId, double cost) {
        connections.put(destinationId, cost);
    }
}
```

**2 Order.java**

```
public class Order {
    public int id;
    public int deliveryPointId;
    public int timeWindowStart; // in minutes (e.g., 0 = now, 30 = in 30 mins)
    public int timeWindowEnd;

    public Order(int id, int deliveryPointId, int start, int end) {
        this.id = id;
        this.deliveryPointId = deliveryPointId;
        this.timeWindowStart = start;
        this.timeWindowEnd = end;
    }
}
```

**3 Rider.java**

```
import java.util.ArrayList;
import java.util.List;

public class Rider {
    public int id;
    public int maxCapacity;
    public List<Order> assignedOrders;

    public Rider(int id, int maxCapacity) {
        this.id = id;
        this.maxCapacity = maxCapacity;
        this.assignedOrders = new ArrayList<>();
    }

    public boolean canTakeOrder() {
        return assignedOrders.size() < maxCapacity;
    }

    public void assignOrder(Order order) {
        if (canTakeOrder()) {
            assignedOrders.add(order);
        }
    }
}
```

**4 Solution.java**

```
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class Solution {
    public List<Chromosome> riderRoutes;
    public double totalFitness;

    public Solution(List<Chromosome> routes) {
        this.riderRoutes = routes;
        this.totalFitness = Double.MAX_VALUE;
    }




 public void calculateTotalFitness(List<DeliveryPoint> graph, int maxOrdersPerRider) {
        double sum = 0;
        Set<Integer> assigned = new HashSet<>();

        for (Chromosome c : riderRoutes) {
            // Penalize duplicate customer assignments
            for (int id : c.deliverySequence) {
                if (assigned.contains(id)) {
                    sum += 500; // Penalty for duplicate delivery
                } else {
                    assigned.add(id);
                }
            }

            if (c.deliverySequence.size() > maxOrdersPerRider) {
                sum += (c.deliverySequence.size() - maxOrdersPerRider) * 100;
            }

            c.calculateFitness(graph);
            sum += c.fitness;
        }

        this.totalFitness = sum;
    }

}
```

**5 Chromosome.java**

```
import java.util.*;

public class Chromosome {
    public List<Integer> deliverySequence;
    public double fitness;

    public Chromosome() {
        this.deliverySequence = new ArrayList<>();
        this.fitness = Double.MAX_VALUE;
    }

    public Chromosome(List<Integer> sequence) {
        this.deliverySequence = new ArrayList<>(sequence);
        this.fitness = Double.MAX_VALUE;
    }

    public void calculateFitness(List<DeliveryPoint> graph) {
        double totalCost = 0.0;
        double penalty = 1000.0;

        for (int i = 0; i < deliverySequence.size() - 1; i++) {
            int fromId = deliverySequence.get(i);
            int toId = deliverySequence.get(i + 1);

            DeliveryPoint from = findPointById(graph, fromId);
            if (from != null && from.connections.containsKey(toId)) {
                totalCost += from.connections.get(toId);
            } else {
                totalCost += penalty;
            }
        }

        this.fitness = totalCost;
    }

    private DeliveryPoint findPointById(List<DeliveryPoint> graph, int id) {
        for (DeliveryPoint dp : graph) {
            if (dp.id == id) return dp;
        }
        return null;
    }
}
```

**6 GraphBuilder.java**

```
import java.util.*;

public class GraphBuilder {
    public static List<DeliveryPoint> buildGraph() {
        List<DeliveryPoint> graph = new ArrayList<>();

        graph.add(new DeliveryPoint(0, "Rider Hub", 0, 0));
        graph.add(new DeliveryPoint(1, "Customer A", 1, 2));
        graph.add(new DeliveryPoint(2, "Customer B", 3, 1));
        graph.add(new DeliveryPoint(3, "Customer C", 4, 3));
        graph.add(new DeliveryPoint(4, "Customer D", 2, 4));
        graph.add(new DeliveryPoint(5, "Customer E", 5, 2));
        graph.add(new DeliveryPoint(6, "Customer F", 6, 3));
        graph.add(new DeliveryPoint(7, "Customer G", 7, 1));
        graph.add(new DeliveryPoint(8, "Customer H", 6, 5));

        connect(graph, 0, 1, 2.0);
        connect(graph, 0, 2, 2.5);
        connect(graph, 0, 3, 3.0);
        connect(graph, 0, 4, 3.5);
        connect(graph, 1, 2, 1.0);
        connect(graph, 1, 5, 2.0);
        connect(graph, 2, 3, 1.2);
        connect(graph, 3, 4, 1.3);
        connect(graph, 4, 5, 2.0);
        connect(graph, 5, 6, 1.5);
        connect(graph, 6, 7, 1.1);
        connect(graph, 7, 8, 1.3);
        connect(graph, 3, 8, 2.7);
        connect(graph, 0, 5, 4.0);
        connect(graph, 0, 6, 5.0);
        connect(graph, 0, 7, 6.0);
        connect(graph, 0, 8, 6.5);

        return graph;
    }

    private static void connect(List<DeliveryPoint> g, int from, int to, double cost) {
        g.get(from).addConnection(to, cost);
        g.get(to).addConnection(from, cost);
    }
}
```

### 7 GeneticAlgorithm.java

```
import java.util.*;

public class GeneticAlgorithm {
    public List<Solution> population;
    private int populationSize;
    private int maxGenerations;
    private double crossoverRate;
    private double mutationRate;

    public GeneticAlgorithm(int populationSize, int maxGenerations, double crossoverRate,
double mutationRate) {
        this.populationSize = populationSize;
        this.maxGenerations = maxGenerations;
        this.crossoverRate = crossoverRate;
        this.mutationRate = mutationRate;
        this.population = new ArrayList<>();
    }


// Initialize population with random rider assignments
    public void initializePopulation(List<Integer> allOrders, int numRiders) {
        Random rand = new Random();

        for (int i = 0; i < populationSize; i++) {
            List<Integer> shuffledOrders = new ArrayList<>(allOrders);
            Collections.shuffle(shuffledOrders);

            List<Chromosome> routes = new ArrayList<>();
            int index = 0;
            for (int r = 0; r < numRiders; r++) {
                List<Integer> riderOrders = new ArrayList<>();
                int limit = shuffledOrders.size() / numRiders;
                for (int j = 0; j < limit && index < shuffledOrders.size(); j++) {
                    riderOrders.add(shuffledOrders.get(index++));
                }
                routes.add(new Chromosome(riderOrders));
            }

            // Remaining orders go to the last rider
            while (index < shuffledOrders.size()) {
                routes.get(numRiders - 1).deliverySequence.add(shuffledOrders.get(index++));
            }
```

```
            population.add(new Solution(routes));
        }
    }

    // Evaluate total fitness for each solution
    public void evaluateFitness(List<DeliveryPoint> graph, int maxOrdersPerRider) {
        for (Solution s : population) {
            s.calculateTotalFitness(graph, maxOrdersPerRider);
        }
    }

    // Evolve population over generations
    public void evolveMulti(List<DeliveryPoint> graph, int maxOrdersPerRider) {
        for (int generation = 0; generation < maxGenerations; generation++) {
            evaluateFitness(graph, maxOrdersPerRider);
            List<Solution> newPopulation = new ArrayList<>();

            while (newPopulation.size() < populationSize) {
                Solution parent1 = selectParent();
                Solution parent2 = selectParent();

                Solution child;
                if (Math.random() < crossoverRate) {
                    child = crossoverMulti(parent1, parent2);
                } else {
                    child = cloneSolution(parent1);
                }

                if (Math.random() < mutationRate) {
                    child = mutateMulti(child);
                }

                child.calculateTotalFitness(graph, maxOrdersPerRider);
                newPopulation.add(child);
            }

            population = newPopulation;
            System.out.println("Generation " + generation + " | Best Fitness: " +
getBestSolution().totalFitness);
        }
    }
```

```
    // Tournament selection
    public Solution selectParent() {
        List<Solution> tournament = new ArrayList<>();
        Random rand = new Random();
        for (int i = 0; i < 3; i++) {
            tournament.add(population.get(rand.nextInt(population.size())));
        }
        tournament.sort(Comparator.comparingDouble(s -> s.totalFitness));
        return cloneSolution(tournament.get(0));
    }

    // Simple crossover: pick each rider's route from either parent
    public Solution crossoverMulti(Solution p1, Solution p2) {
        Random rand = new Random();
        List<Chromosome> newRoutes = new ArrayList<>();

        for (int i = 0; i < p1.riderRoutes.size(); i++) {
            Chromosome selected = rand.nextBoolean() ? p1.riderRoutes.get(i) :
p2.riderRoutes.get(i);
            newRoutes.add(new Chromosome(new ArrayList<>(selected.deliverySequence))); //
copy
        }

        return new Solution(newRoutes);
    }

    // Mutate each rider route by swapping 2 locations (chance per rider)
    public Solution mutateMulti(Solution s) {
        Random rand = new Random();
        List<Chromosome> newRoutes = new ArrayList<>();

        for (Chromosome route : s.riderRoutes) {
            List<Integer> seq = new ArrayList<>(route.deliverySequence);
            if (seq.size() >= 2 && rand.nextDouble() < 0.5) {
                int i = rand.nextInt(seq.size());
                int j = rand.nextInt(seq.size());
                Collections.swap(seq, i, j);
            }
            newRoutes.add(new Chromosome(seq));
        }

        return new Solution(newRoutes);
    }
```

```
    // Best solution in current population
    public Solution getBestSolution() {
        return Collections.min(population, Comparator.comparingDouble(s -> s.totalFitness));
    }

    // Helper to clone a solution
    public Solution cloneSolution(Solution original) {
        List<Chromosome> newRoutes = new ArrayList<>();
        for (Chromosome c : original.riderRoutes) {
            newRoutes.add(new Chromosome(new ArrayList<>(c.deliverySequence)));
        }
        return new Solution(newRoutes);
    }
}
```

## 8 GeneticAlgorithmOptimizer.java

```
import java.util.*;
public class GeneticAlgorithmOptimizer {

    public static class Result {
        public Map<Integer, List<Order>> routes;
        public double totalFitness;

        public Result(Map<Integer, List<Order>> routes, double fitness) {
            this.routes = routes;
            this.totalFitness = fitness;
        }
    }

    public static Result optimize(List<DeliveryPoint> graph, List<Order> orders, List<Rider>
originalRiders, int maxGenerations) {
        List<Map<Integer, List<Order>>> population = new ArrayList<>();
        Random random = new Random();

        // Initialize population with random assignments
        for (int i = 0; i < 10; i++) {
            population.add(generateRandomAssignment(orders, originalRiders));
        }

        Map<Integer, List<Order>> bestIndividual = null;
        double bestFitness = Double.MAX_VALUE;

        for (int generation = 0; generation < maxGenerations; generation++) {
            for (Map<Integer, List<Order>> individual : population) {
                double fitness = calculateFitness(individual, graph);

                if (fitness < bestFitness) {
                    bestFitness = fitness;
                    bestIndividual = deepCopy(individual);
                }
            }

            // Simple mutation logic (optional: implement crossover)
            population = mutatePopulation(population, orders, originalRiders);
        }

        return new Result(bestIndividual, bestFitness);
    }
```

```
    private static Map<Integer, List<Order>> generateRandomAssignment(List<Order> orders,
List<Rider> originalRiders) {
        Map<Integer, List<Order>> routes = new HashMap<>();
        Random random = new Random();

        // Reset riders
        List<Rider> riders = new ArrayList<>();
        for (Rider r : originalRiders) {
            riders.add(new Rider(r.id, r.maxCapacity));
        }

        for (Order order : orders) {
            Rider chosen = null;
            int attempts = 0;
            while (attempts < 10) {
                Rider candidate = riders.get(random.nextInt(riders.size()));
                if (candidate.canTakeOrder()) {
                    chosen = candidate;
                    break;
                }
                attempts++;
            }

            if (chosen != null) {
                chosen.assignOrder(order);
                routes.computeIfAbsent(chosen.id, k -> new ArrayList<>()).add(order);
            }
        }

        return routes;
    }

    private static double calculateFitness(Map<Integer, List<Order>> routes, List<DeliveryPoint>
graph) {
        double total = 0;
        for (Map.Entry<Integer, List<Order>> entry : routes.entrySet()) {
            int fromId = 0; // start at hub
            for (Order order : entry.getValue()) {
                int toId = order.deliveryPointId;
                DeliveryPoint from = findPointById(graph, fromId);
                if (from != null && from.connections.containsKey(toId)) {
                    total += from.connections.get(toId);
                    fromId = toId;
                } else {
```

```
                total += 9999; // unreachable penalty
            }
        }
    }
    return total;
}

private static DeliveryPoint findPointById(List<DeliveryPoint> graph, int id) {
    for (DeliveryPoint dp : graph) {
        if (dp.id == id) return dp;
    }
    return null;
}

private static Map<Integer, List<Order>> deepCopy(Map<Integer, List<Order>> original) {
    Map<Integer, List<Order>> copy = new HashMap<>();
    for (Map.Entry<Integer, List<Order>> entry : original.entrySet()) {
        copy.put(entry.getKey(), new ArrayList<>(entry.getValue()));
    }
    return copy;
}

private static List<Map<Integer, List<Order>>> mutatePopulation(List<Map<Integer,
List<Order>>> population, List<Order> orders, List<Rider> riders) {
    List<Map<Integer, List<Order>>> mutated = new ArrayList<>();
    for (int i = 0; i < population.size(); i++) {
        mutated.add(generateRandomAssignment(orders, riders));  // simple re-randomization
    }
    return mutated;
}
}
```

**9 FastBiteOptimizer.java**

```
import java.util.*;

public class FastBiteOptimizer {
    public static void main(String[] args) {
        List<DeliveryPoint> graph = GraphBuilder.buildGraph();

        List<Integer> allOrders = new ArrayList<>();
        for (DeliveryPoint dp : graph) {
            if (dp.id != 0) allOrders.add(dp.id); // 0 is Rider Hub
        }

        // GA Parameters
        int populationSize = 50;
        int maxGenerations = 100;
        double crossoverRate = 0.8;
        double mutationRate = 0.3;
        int numRiders = 3;
        int maxOrdersPerRider = 4;

        // 🕒 Start timing
        long startTime = System.nanoTime();

        GeneticAlgorithm ga = new GeneticAlgorithm(populationSize, maxGenerations,
crossoverRate, mutationRate);
        ga.initializePopulation(allOrders, numRiders);
        ga.evolveMulti(graph, maxOrdersPerRider);
        Solution best = ga.getBestSolution();

        // 🕒 End timing
        long endTime = System.nanoTime();
        double durationMs = (endTime - startTime) / 1_000_000.0;

        // ✅ Output results
        System.out.println("\n📦 Genetic Algorithm – Best Multi-Rider Delivery Plan:");
        for (int i = 0; i < best.riderRoutes.size(); i++) {
            System.out.println("Rider " + (i + 1) + ": " + best.riderRoutes.get(i).deliverySequence);
        }
        System.out.printf("🧮 Total Delivery Cost (Fitness): %.2f\n", best.totalFitness);
        System.out.printf("🕒 GA Execution Time: %.2f ms\n", durationMs);
    }
}
```

**10 UnitTest.java**

```
import java.util.*;

public class UnitTest {

    public static void main(String[] args) {
        testGeneticAlgorithm();
        testSortingBasedAssignment();
    }

    private static void testGeneticAlgorithm() {
        System.out.println("\n🧪 Running Unit Test: Genetic Algorithm");

        List<DeliveryPoint> graph = GraphBuilder.buildGraph();
        List<Order> orders = generateTestOrders();
        List<Rider> riders = generateTestRiders();

        GeneticAlgorithmOptimizer.Result result = GeneticAlgorithmOptimizer.optimize(graph,
orders, riders, 10);

        System.out.println("➡️ Checking GA result object...");
        assert result != null : "❌ GA Result should not be null";
        System.out.println("✅ GA result object is not null");

        assert result.routes != null : "❌ GA Routes should not be null";
        System.out.println("✅ GA routes are not null");

        assert result.totalFitness >= 0 : "❌ GA Fitness should be non-negative";
        System.out.println("✅ GA fitness is non-negative: " + result.totalFitness);

        System.out.println("➡️ Checking for duplicate orders in GA...");
        checkNoDuplicates(result.routes);

        System.out.println("➡️ Checking capacity limits for riders in GA...");
        checkCapacityLimits(result.routes, 4);

        System.out.println("✅ All Genetic Algorithm tests passed!");
    }

    private static void testSortingBasedAssignment() {
        System.out.println("\n🧪 Running Unit Test: Sorting-Based Assignment");

        List<DeliveryPoint> graph = GraphBuilder.buildGraph();
        List<Order> orders = generateTestOrders();
```

```
        List<Rider> riders = generateTestRiders();

        SortingBasedAssignment.AssignmentResult result =
SortingBasedAssignment.assignOrders(graph, orders, riders);

        System.out.println("➡️ Checking SBA result object...");
        assert result != null : "❌ SBA Result should not be null";
        System.out.println("✅ SBA result object is not null");

        assert result.assignedRoutes != null : "❌ SBA Routes should not be null";
        System.out.println("✅ SBA routes are not null");

        assert result.totalCost >= 0 : "❌ SBA Cost should be non-negative";
        System.out.println("✅ SBA cost is non-negative: " + result.totalCost);

        System.out.println("➡️ Checking for duplicate orders in SBA...");
        checkNoDuplicates(result.assignedRoutes);

        System.out.println("➡️ Checking capacity limits for riders in SBA...");
        checkCapacityLimits(result.assignedRoutes, 4);

        System.out.println("✅ All Sorting-Based Assignment tests passed!");
    }

    private static void checkNoDuplicates(Map<Integer, List<Order>> routes) {
        Set<Integer> seen = new HashSet<>();
        for (List<Order> orders : routes.values()) {
            for (Order o : orders) {
                assert !seen.contains(o.id) : "❌ Duplicate order found: " + o.id;
                seen.add(o.id);
            }
        }
        System.out.println("✅ No duplicate orders found.");
    }

    private static void checkCapacityLimits(Map<Integer, List<Order>> routes, int maxCapacity) {
        for (Map.Entry<Integer, List<Order>> entry : routes.entrySet()) {
            assert entry.getValue().size() <= maxCapacity :
                "❌ Rider " + entry.getKey() + " exceeded capacity.";
        }
        System.out.println("✅ All riders are within capacity limits.");
    }

    private static List<Order> generateTestOrders() {
```

```
        return new ArrayList<>(List.of(
            new Order(1, 1, 0, 30),
            new Order(2, 2, 0, 30),
            new Order(3, 3, 0, 30),
            new Order(4, 4, 0, 30),
            new Order(5, 5, 0, 30),
            new Order(6, 6, 0, 30),
            new Order(7, 7, 0, 30),
            new Order(8, 8, 0, 30)
        ));
    }

    private static List<Rider> generateTestRiders() {
        return new ArrayList<>(List.of(
            new Rider(1, 4),
            new Rider(2, 4),
            new Rider(3, 4)
        ));
    }
}
```

**11 SortingBasedAlgorithm.java**

```
import java.util.*;

public class SortingBasedAssignment {

    private static final double UNREACHABLE_PENALTY = 9999;
    private static final double UNASSIGNED_ORDER_PENALTY = 1000;

    public static class AssignmentResult {
        public Map<Integer, List<Order>> assignedRoutes;
        public double totalCost;

        public AssignmentResult(Map<Integer, List<Order>> assignedRoutes, double totalCost) {
            this.assignedRoutes = assignedRoutes;
            this.totalCost = totalCost;
        }
    }

    public static AssignmentResult assignOrders(List<DeliveryPoint> graph, List<Order> orders,
List<Rider> riders) {
        orders.sort(Comparator.comparingInt(o -> o.timeWindowStart));
        Map<Integer, List<Order>> assignedRoutes = new HashMap<>();
        double totalCost = 0;

        for (Order order : orders) {
            Rider bestRider = null;
            double minCost = Double.MAX_VALUE;

            List<Rider> shuffledRiders = new ArrayList<>(riders);
            Collections.shuffle(shuffledRiders);

            for (Rider rider : shuffledRiders) {
                if (rider.canTakeOrder()) {
                    int fromId = 0;
                    int toId = order.deliveryPointId;
                    DeliveryPoint from = findPointById(graph, fromId);
                    double cost = (from != null && from.connections.containsKey(toId)) ?
                            from.connections.get(toId) : UNREACHABLE_PENALTY;

                    if (cost < minCost) {
                        minCost = cost;
                        bestRider = rider;
                    }
                }
```

```
        }

        if (bestRider != null && minCost < UNREACHABLE_PENALTY) {
            bestRider.assignOrder(order);
            assignedRoutes.computeIfAbsent(bestRider.id, k -> new ArrayList<>()).add(order);
            totalCost += minCost;
        } else {
            totalCost += UNASSIGNED_ORDER_PENALTY;
        }
    }

    return new AssignmentResult(assignedRoutes, totalCost);
}

private static DeliveryPoint findPointById(List<DeliveryPoint> graph, int id) {
    for (DeliveryPoint dp : graph) {
        if (dp.id == id) return dp;
    }
    return null;
}

public static void main(String[] args) {
    List<DeliveryPoint> graph = GraphBuilder.buildGraph();

    List<Order> orders = new ArrayList<>(List.of(
        new Order(1, 1, 0, 30),
        new Order(2, 2, 0, 30),
        new Order(3, 3, 0, 30),
        new Order(4, 4, 0, 30),
        new Order(5, 5, 0, 30),
        new Order(6, 6, 0, 30),
        new Order(7, 7, 0, 30),
        new Order(8, 8, 0, 30)
    ));

    List<Rider> riders = new ArrayList<>(List.of(
        new Rider(1, 4),
        new Rider(2, 4),
        new Rider(3, 4)
    ));

    long start = System.nanoTime();
    AssignmentResult result = assignOrders(graph, orders, riders);
    long end = System.nanoTime();
```

```
        double duration = (end - start) / 1_000_000.0;

        System.out.println("\n📦 Sorting-Based Delivery Assignment (Enhanced):");
        for (Rider rider : riders) {
            List<Order> assigned = result.assignedRoutes.getOrDefault(rider.id, new ArrayList<>());
            System.out.print("Rider " + rider.id + ": ");
            for (Order o : assigned) {
                System.out.print("Order" + o.id + " ");
            }
            if (assigned.isEmpty()) {
                System.out.print("[No orders assigned]");
            }
            System.out.println();
        }

        System.out.printf("🧮 Total Delivery Cost (Enhanced Fitness): %.2f\n", result.totalCost);
        System.out.printf("🕒 SBA Execution Time: %.2f ms\n", duration);
    }
}
```

## 11.0 Appendix B

**Initial Project Plan (week 10, submission date: 31 May 2024)**

<table>
<tr><td>Group Name</td><td>1</td></tr>
<tr><td>Members</td><td>
<table>
<tr><th>Name</th><th>Email</th><th>Phone number</th></tr>
<tr><td>Muhammad Hazlam Azwar bin Mohd Farizal</td><td>215838@student.upm.edu.my</td><td>+601119546043</td></tr>
<tr><td>Abdul Majid bin Mirsruji</td><td>214727@student.upm.edu.my</td><td>+60183949296</td></tr>
<tr><td>Muhammad Faiz bin Md Fadzli</td><td>215768@student.upm.edu.my</td><td>+60132916220</td></tr>
<tr><td>Muhammad Akmal bin Wanahari</td><td>215666@student.upm.edu.my</td><td>+60136008690</td></tr>
<tr><td>Muhammad Aqil Iqbal bin Jamil</td><td>215728@student.upm.edu.my</td><td>+60196829171</td></tr>
</table>
</td></tr>
<tr><td>Problem scenario description</td><td>FastBite, a food delivery platform, aims to optimize its multi-rider delivery system across a city. With increasing orders, traffic congestion, and rider capacity limits, it needs to assign orders efficiently to minimize total delivery cost while meeting time windows and avoiding overload.</td></tr>
<tr><td>Why it is important</td><td>Inefficient delivery assignment leads to late deliveries, wasted fuel, and poor customer satisfaction. An optimized system improves operational cost, customer experience, and scalability — critical in high-demand urban areas.</td></tr>
<tr><td>Problem specification</td><td>● Input: Graph of delivery points, rider list, orders with location & time windows<br>● Constraints: Rider capacity, no duplicate deliveries, cost based on distance & time<br>● Output: Optimized multi-rider delivery plan with minimized total delivery cost</td></tr>
<tr><td>Potential solutions</td><td>● Greedy: Fast but ignores global optimization<br>● Sorting-based: Simple and fast but inefficient under constraints</td></tr>
</table>

| | |
|---|---|
| | • Graph + Genetic Algorithm (chosen): Adaptive, handles constraints, offers near-optimal solutions through evolutionary search |
| Sketch (framework, flow, interface) | i)Flow<br>• Build graph →<br>• Generate population (GA) or sort orders (SBA) →<br>• Assign orders respecting constraints →<br>• Calculate total cost →<br>• Output best plan<br>ii)Interface<br>Console-based output; Portfolio on GitHub Pages with visual diagrams and result screenshots |

**Project Proposal Refinement (week 11, submission date: 7 June 2023)**

<table>
<tr><td>Group Name</td><td>1</td></tr>
<tr><td>Members</td><td>
<table>
<tr><th>Name</th><th>Role</th></tr>
<tr><td>Muhammad Hazlam Azwar bin Mohd Farizal</td><td>Java Programmer</td></tr>
<tr><td>Abdul Majid bin Mirsruji</td><td>Algorithm Designer</td></tr>
<tr><td>Muhammad Faiz bin<br>Md Fadzli</td><td>Analyzer and portfolio builder</td></tr>
<tr><td>Muhammad Akmal bin Wanahari</td><td>Writer</td></tr>
<tr><td>Muhammad Aqil Iqbal bin Jamil</td><td>Algorithm Researcher</td></tr>
</table>
</td></tr>
<tr><td>Problem statement</td><td>FastBite, a food delivery service, struggles to assign multiple customer orders to a limited number of riders efficiently. Constraints such as delivery time windows, traffic, and rider capacity make manual or naive assignment strategies ineffective.</td></tr>
<tr><td>Objectives</td><td>● Minimize total delivery cost (time, distance, fuel)<br>● Assign all orders without exceeding rider capacity<br>● Ensure timely delivery while avoiding duplication</td></tr>
<tr><td>Expected output</td><td>● A multi-rider delivery plan that minimizes cost<br>● Rider-to-order assignment respecting constraints<br>● Execution time and delivery cost for each approach</td></tr>
<tr><td>Problem scenario description</td><td>In a growing city, FastBite faces logistical issues in delivering food orders with limited riders during peak hours. Routes overlap, and some riders are overburdened while others are underutilized. A smarter routing and assignment algorithm is needed to handle increasing demand and ensure efficiency.</td></tr>
<tr><td>Why it is important</td><td>Optimized delivery enhances customer satisfaction, reduces delivery time, saves fuel, and balances rider workload. This is crucial for business scalability, cost efficiency, and real-time logistics management in a competitive food delivery industry.</td></tr>
</table>

| | |
|---|---|
| Problem specification | i)Inputs:<br>● Graph of delivery points (nodes, edges with costs)<br>● Orders with time windows and locations<br>● Rider list with max order capacity<br><br>ii)Constraints:<br>● Each rider can only carry a limited number of orders<br>● No duplicate delivery assignments<br>● Orders must be assigned within time and cost limits<br><br>iii)Output:<br>● Valid assignment of orders to riders<br>● Total delivery cost<br>● Order distribution balance |
| Potential solutions | ● Greedy Algorithm: Fast but locally optimal only<br>● Sorting-Based Assignment: Simple and fast but lacks constraint awareness<br>● Graph + Genetic Algorithm (Selected): Evolves valid routes using crossover, mutation, and fitness evaluation to find cost-effective and feasible multi-rider delivery solutions |
| Sketch (framework, flow, interface) | i)Flow<br>● Build graph →<br>● Generate population (GA) or sort orders (SBA) →<br>● Assign orders respecting constraints →<br>● Calculate total cost →<br>● Output best plan<br>ii)Interface<br>Console-based output; Portfolio on GitHub Pages with visual diagrams and result screenshots |

| | |
|---|---|
| Methodology | |

| Milestone | Time |
|---|---|
| <eg: scenario refinement> | wk10 |
| <eg: find example solutions and suitable algorithm. Discuss in group why that solution and the example problems relate to the problem in the project> | wk11 |
| <eg: edit the coding of the chosen problem and complete the coding. Debug> | wk12 |
| <eg: conduct analysis of correctness and time complexity > | wk13 |
| <prepare online portfolio and presentation> | wk14 |

**Project Progress (Week 10 – Week 14)**

<table>
<tr><td>Milestone 1</td><td></td></tr>
<tr><td>Date (week)</td><td>Week 10</td></tr>
<tr><td>Description/ sketch</td><td>• Defined problem: multi-rider delivery assignment for FastBite<br>• Identified key constraints (rider capacity, distance, time)<br>• Discussed algorithm choices and selected GA & SBA<br>• Assigned roles and set project timeline</td></tr>
<tr><td>Role</td><td>
<table>
<tr><td>Task</td><td>Hazlam (Java Programmer)</td><td>Majid (Algortihm Designer)</td><td>Faiz (Analyzer & Portfolio)</td><td>Akmal (Writer)</td><td>Aqil (Researcher)</td></tr>
<tr><td>Problem modeling</td><td>✅</td><td>✅</td><td>✅</td><td></td><td>✅</td></tr>
<tr><td>Planning and scheduling</td><td>✅</td><td></td><td>✅</td><td>✅</td><td>✅</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td>Milestone 2</td><td></td></tr>
<tr><td>Date (Wk)</td><td>Week 11</td></tr>
<tr><td>Description/ sketch</td><td>• Compared algorithms and finalized GA & SBA<br>• Started Java coding: order, rider, graph classes<br>• Defined fitness and penalty logic<br>• Researched evolutionary optimization techniques</td></tr>
<tr><td>Role</td><td>
<table>
<tr><td>Task</td><td>Hazlam</td><td>Majid</td><td>Faiz</td><td>Akmal</td><td>Aqil</td></tr>
<tr><td>Java coding started</td><td>✅</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Algorithm design</td><td></td><td>✅</td><td></td><td></td><td>✅</td></tr>
<tr><td>Research & comparison</td><td></td><td>✅</td><td></td><td></td><td>✅</td></tr>
</table>
</td></tr>
</table>

| | | | | | | |
|---|---|---|---|---|---|---|
| | Documentation draft | | | | ✅ | |

| | |
|---|---|
| **Milestone 3** | |
| **Date (Wk)** | Week 12 |
| **Description/ sketch** | • Completed core implementation and unit tests<br>• Conducted rider capacity validation and duplicate checks<br>• Compared GA and SBA on sample graph (8 orders, 3 riders)<br>• Refined Java structure for modularity |
| **Role** | |

| Task | Hazlam | Majid | Faiz | Akmal | Aqil |
|---|---|---|---|---|---|
| Java debugging/tests | ✅ | | | | |
| Validation logic | ✅ | | ✅ | | |
| Result analysis | | ✅ | ✅ | | |
| Documentation update | | | | ✅ | |

<table>
<tr><td>Milestone 4</td><td></td></tr>
<tr><td>Date (Wk)</td><td>Week 13</td></tr>
<tr><td>Description/ sketch</td><td>
<ul>
<li>Verified correctness of GA using test cases</li>
<li>Analyzed time complexity for both methods</li>
<li>Confirmed performance difference between GA and SBA</li>
<li>Documented edge cases and theoretical justification</li>
</ul>
</td></tr>
<tr><td>Role</td><td>
<table>
<tr><th>Task</th><th>Hazlam</th><th>Majid</th><th>Faiz</th><th>Akmal</th><th>Aqil</th></tr>
<tr><td>Time complexity analysis</td><td></td><td>✅</td><td>✅</td><td></td><td></td></tr>
<tr><td>Test case validation</td><td>✅</td><td></td><td>✅</td><td></td><td></td></tr>
<tr><td>Edge case documentation</td><td></td><td></td><td>✅</td><td>✅</td><td></td></tr>
<tr><td>Theory & correctness write-up</td><td></td><td>✅</td><td></td><td>✅</td><td>✅</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td>Milestone 5</td><td></td></tr>
<tr><td>Date (Wk)</td><td>Week 14</td></tr>
<tr><td>Description/ sketch</td><td>
<ul>
<li>Finalized GitHub Pages portfolio</li>
<li>Created and rehearsed slides for class presentation</li>
<li>Uploaded Java source code, documentation, and sample outputs</li>
<li>Prepared final submission zip with all required materials</li>
</ul>
</td></tr>
<tr><td>Role</td><td>
<table>
<tr><th>Task</th><th>Hazlam</th><th>Majid</th><th>Faiz</th><th>Akmal</th><th>Aqil</th></tr>
<tr><td>Slide preparation</td><td>✅</td><td>✅</td><td>✅</td><td>✅</td><td>✅</td></tr>
<tr><td>Final portfolio upload</td><td></td><td></td><td>✅</td><td></td><td></td></tr>
<tr><td>Presentation rehearsal</td><td>✅</td><td>✅</td><td>✅</td><td>✅</td><td>✅</td></tr>
<tr><td>Final code</td><td>✅</td><td></td><td></td><td></td><td></td></tr>
</table>
</td></tr>
</table>